**Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Departamentul Informatică și Ingineria Sistemelor**

**RAPORT**

**Proiect de an la TMPS**

A efectuat:

st. gr. TI-203 Chiforiuc Adrian

A verificat:

l. univ. Mihai Gaidau

Chişinău – 2023

Contents

[INTRODUCERA 3](#_Toc136588721)

[DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE 5](#_Toc136588723)

[DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU 7](#_Toc136588724)

[MODELAREA SISTEMULUI UML 11](#_Toc136588725)

[CONCLUZIE 15](#_Toc136588726)

[BIBLIOGRAFIE 17](#_Toc136588727)

# INTRODUCERA

Design patternurile reprezintă soluții recurente la problemele comune în designul și implementarea software-ului. Acestea sunt abordări testate și validate de experți în domeniul programării și reprezintă o modalitate eficientă de a rezolva problemele de proiectare într-un mod flexibil și reutilizabil.

Unul dintre cele mai cunoscute design patternuri este "Singleton". Acesta se referă la crearea unei clase care permite existența unei singure instanțe a sa și oferă un punct centralizat de acces la această instanță în întreaga aplicație. Singleton este folosit atunci când avem nevoie de o singură instanță a unei clase, cum ar fi o bază de date sau un obiect de configurare.

Un alt design pattern important este "Factory". Acesta se referă la crearea unui obiect prin intermediul unei metode de fabrică, fără a dezvălui logica exactă de creare a obiectului în clientul care solicită crearea acestuia. Astfel, Factory Pattern ne permite să abstractizăm procesul de creare a obiectelor și să obținem o flexibilitate sporită în schimbul tipului de obiect creat.

Mai există și design patternul "Observer", care definește o relație de tip "unul-la-mulți" între obiecte, astfel încât atunci când unul dintre obiecte se modifică, toți observatorii să fie notificați și să poată reacționa corespunzător. Acest pattern este folosit frecvent în dezvoltarea interfețelor grafice sau în sistemele distribuite, unde schimbările trebuie propagate în întreaga aplicație.

Un alt design pattern important este "Decorator". Acesta permite adăugarea de funcționalități suplimentare la un obiect existent, fără a afecta alte instanțe ale aceleiași clase. Decoratorul ne permite să extindem comportamentul unui obiect într-un mod flexibil, adăugând și eliminând funcționalități la runtime.

Patternul Composite ne ajută să construim structuri de obiecte într-o ierarhie arborescentă, astfel încât obiectele individuale sau grupuri de obiecte să fie tratate în mod uniform. În acest pattern, fiecare obiect din ierarhie poate fi tratat atât ca o entitate individuală, cât și ca parte a unei colecții mai mari. Acest lucru permite compunerea și manipularea obiectelor într-un mod recursiv și flexibil. De exemplu, putem crea o ierarhie de componente UI, în care o componentă poate conține alte componente sau obiecte individuale, oferind astfel o modalitate convenabilă de a crea structuri complexe și ușor de gestionat.

Patternul Chain of Responsibility permite construirea unei lanțuri de obiecte, fiecare având posibilitatea de a trata sau a transmite solicitările către următorul obiect din lanț. Acest pattern ne permite să separăm sender-ul de receiver-i și să construim un flux de procesare al cererilor în mod dinamic. Astfel, fiecare obiect din lanț decide dacă poate trata cererea sau o transmite mai departe. De exemplu, într-un sistem de gestionare a evenimentelor, avem mai mulți handler-i care se ocupă de diferite tipuri de evenimente. Fiecare handler decide dacă poate trata evenimentul sau îl trece către handler-ul următor din lanț până când evenimentul este procesat sau ajunge la sfârșitul lanțului.

Design patternurile sunt o unealtă valoroasă în cadrul dezvoltării software-ului, deoarece ele oferă o soluție standardizată la problemele frecvente de proiectare. Utilizarea acestor patternuri duce la crearea unui cod mai modular, mai flexibil și mai ușor de întreținut. Cu toate acestea, este important să se ia în considerare și contextul specific al fiecărei aplicații și să se aplice patternurile în mod rațional și adaptat la nevoile proiectului.

# DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE

Aplicația de gestionare a contactelor telefonice este un proiect Java care utilizează diverse modele de design pentru a oferi utilizatorilor o modalitate eficientă de organizare și administrare a contactelor lor telefonice. Aceste modele de design, cum ar fi Singleton, Builder, Factory Method, Adapter, Composite, Decorator, Chain of Responsibility și Strategy, au fost implementate în aplicație pentru a asigura o structură modulară, ușor de întreținut și extinsă.

Aplicația de gestionare a contactelor telefonice își propune să ofere o experiență intuitivă și funcționalități avansate pentru utilizatori în gestionarea contactelor lor telefonice. Implementarea modelelor de design menționate anterior contribuie la realizarea acestui obiectiv prin următoarele moduri:

1. Singleton:

* A fost utilizat în implementarea clasei de gestionare a contactelor pentru a asigura că există o singură instanță a acesteia în cadrul aplicației.
* Aceasta permite accesul centralizat la funcționalitățile de gestionare a contactelor și evită crearea de instanțe multiple și redundante.

1. Builder:

* A fost utilizat în procesul de creare a obiectelor de contact.
* Acest design pattern facilitează construirea obiectelor complexe pas cu pas, permițând setarea selectivă a atributelor și asigurând o construcție coerentă și controlată.

1. Factory Method:

* A fost utilizat pentru crearea instanțelor de contacte.
* Acest design pattern abstractizează procesul de creare a obiectelor și permite subclaselor să decidă tipul exact de obiect de contact pe care îl creează.

1. Adapter:

* A fost utilizat pentru a adapta clasa de contact la interfața necesară în aplicație.
* Prin intermediul acestui design pattern, clasa de contact existentă poate fi utilizată într-un mod transparent și compatibil cu restul aplicației.

1. Composite:

* A fost utilizat în implementarea grupurilor de contacte.
* Acest design pattern permite tratarea uniformă a obiectelor individuale și a grupurilor de obiecte, permițând utilizatorilor să opereze asupra acestora în mod recursiv și modular.

1. Decorator:

* A fost utilizat pentru a adăuga funcționalitate suplimentară la obiectele de contact existente.
* Prin intermediul acestui design pattern, pot fi adăugate caracteristici suplimentare, cum ar fi atașarea imaginilor de profil, fără a afecta structura de bază a obiectelor de contact.

1. Chain of Responsibility:

* A fost utilizat pentru a gestiona în mod flexibil și modular operațiunile de sortare a contactelor.
* Acest design pattern permite definirea unei serii de obiecte care pot prelucra o cerere într-o manieră secvențială și opțională.

1. Strategy:

* A fost utilizat pentru a permite utilizatorilor să aleagă strategii de sortare personalizate pentru contacte.
* Acest design pattern oferă o abordare flexibilă în selectarea și aplicarea algoritmilor de sortare pe baza preferințelor utilizatorilor.

Aceste modele de design au fost implementate în aplicația de gestionare a contactelor telefonice pentru a asigura o structură modulară, extensibilă și ușor de întreținut. Ele contribuie la separarea responsabilităților, reducerea dependențelor și oferirea de funcționalități avansate și flexibile pentru utilizatori.

# DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU

Aplicația de gestionare a contactelor telefonice este concepută pentru a oferi utilizatorilor o modalitate eficientă de organizare și administrare a contactelor lor într-un mod ușor și intuitiv. Principiul de lucru al aplicației se bazează pe utilizarea unor modele de design renumite, care facilitează structura modulară, extensibilitatea și flexibilitatea acesteia.

La nivelul design-ului arhitectural, aplicația utilizează principiul Singleton pentru a asigura existența unei singure instanțe a clasei ContactManager, care gestionează operațiunile legate de contacte. Acest design pattern permite accesul centralizat la funcționalitățile de gestionare a contactelor și evită crearea de instanțe redundante.

Pentru crearea și configurarea obiectelor de contact, aplicația folosește design pattern-ul Builder. Clasa ContactBuilder oferă metode pentru setarea selectivă a atributelor unui contact și returnează un obiect de contact final construit. Acesta permite crearea de obiecte complexe într-un mod controlat și modular.

În ceea ce privește adaptarea clasei de contact existente la interfața necesară în aplicație, aplicația folosește design pattern-ul Adapter. Clasa ContactAdapterImpl este responsabilă de adaptarea clasei Contact existente la interfața ContactAdapter, astfel încât să poată fi utilizată în cadrul aplicației fără a afecta funcționalitatea existentă.

Pentru a gestiona operațiunile de sortare a contactelor, aplicația utilizează design pattern-ul Chain of Responsibility. Clasa ContactSorter definește un lanț de obiecte SortHandler care prelucrează în mod secvențial solicitările de sortare. Fiecare SortHandler din lanț verifică dacă poate prelucra solicitarea și, în caz afirmativ, sortează contactele conform criteriilor specifice. Pentru a adăuga funcționalități suplimentare la obiectele de contact existente, aplicația utilizează design pattern-ul Decorator. Clasa ContactDecorator este o clasă abstractă care extinde ContactAdapterImpl și permite adăugarea de funcționalități suplimentare la obiectele de contact. De exemplu, clasa ContactWithPhotoDecorator adaugă funcționalitatea de atașare a unei imagini de profil la un contact existent.

De asemenea, aplicația oferă utilizatorilor posibilitatea de a alege strategii personalizate de sortare a contactelor prin intermediul design pattern-ului Strategy. Clasa ContactSortStrategy reprezintă contextul care utilizează o strategie de sortare specifică implementată de clasele derivate ale interfeței SortStrategy. Această abordare permite utilizatorilor să definească și să aplice algoritmi de sortare specifici preferințelor lor.

Prin implementarea acestor modele de design, aplicația de gestionare a contactelor telefonice oferă o structură modulară, ușor de întreținut și de extins. Aceasta separă responsabilitățile și permite adăugarea și configurarea flexibilă a funcționalităților în funcție de nevoile utilizatorilor. Utilizând aceste modele de design, aplicația promovează o abordare coerentă și eficientă în gestionarea și organizarea contactelor telefonice.

Pentru a înțelege cum lucrează aplicația de gestionare a contactelor telefonice, să ne imaginăm următorul exemplu: Utilizatorul deschide aplicația și este prezentat cu o interfață grafică intuitivă. Acesta poate vizualiza lista de contacte existente și poate interacționa cu aceasta prin diverse funcționalități.

* + - 1. Adăugarea unui contact:

Utilizatorul dorește să adauge un contact nou în agenda sa. Folosind interfața grafică, acesta introduce numele, numărul de telefon și alte informații relevante despre contactul nou. Apoi, apasă butonul "Adăugare contact". În spatele ecranului, design pattern-ul Builder intră în acțiune. Clasa ContactBuilder primește informațiile introduse de utilizator și creează un obiect Contact cu aceste detalii. Astfel, un nou contact este adăugat la lista de contacte.

* + - 1. Sortarea contactelor:

Utilizatorul dorește să sorteze lista de contacte în ordine alfabetică după nume. Acesta apasă butonul "Sortare după nume". La nivelul codului, design pattern-ul Chain of Responsibility intră în acțiune. Clasa ContactSorter preia solicitarea de sortare și o trimite către primul obiect din lanțul de SortHandler-i. Fiecare SortHandler verifică dacă poate procesa solicitarea și, în caz afirmativ, sortează lista de contacte conform criteriilor respective. Astfel, contactele sunt reordonate în ordine alfabetică după nume.

* + - 1. Adăugarea unei fotografii de profil:

Utilizatorul dorește să adauge o fotografie de profil unui contact existent. Acesta selectează contactul în interfața grafică și apasă butonul "Adăugare fotografie". Design pattern-ul Decorator intră în acțiune în acest caz. Clasa ContactWithPhotoDecorator este asociată cu contactul selectat și adaugă funcționalitatea de atașare a unei imagini de profil la acesta. Astfel, fotografia de profil este asociată cu contactul selectat și este afișată în interfață.

* + - 1. Selecția unei strategii de sortare personalizate:

Utilizatorul dorește să sorteze contactele după numărul de telefon. Acesta selectează opțiunea "Sortare după număr de telefon" și apasă butonul corespunzător. Design pattern-ul Strategy intră în acțiune în acest caz. Clasa ContactSortStrategy este responsabilă de selectarea și utilizarea strategiei de sortare specificată de utilizator. Această strategie specifică implementează interfața SortStrategy și conține algoritmul de sortare după numărul de telefon. Astfel, lista de contacte este sortată în funcție de numărul de telefon.

Acestea sunt doar câteva exemple de interacțiuni și funcționalități ale aplicației de gestionare a contactelor telefonice. Design pattern-urile implementate asigură o structură modulară, ușor de întreținut și de extins, oferind utilizatorilor o experiență simplă și eficientă în gestionarea contactelor lor.

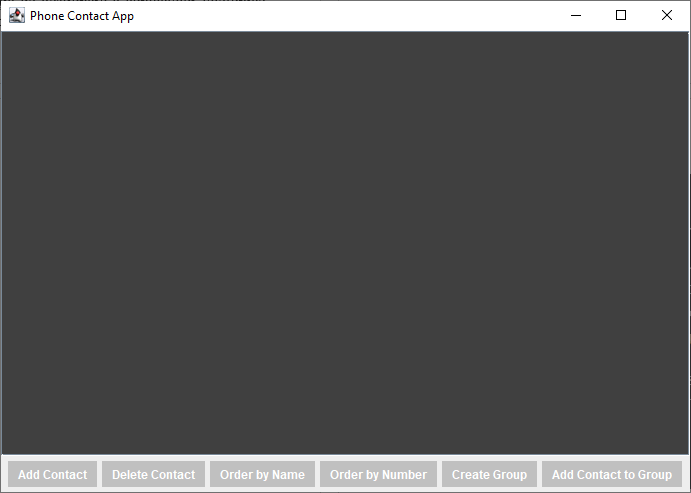


Figura 1.1 Interfața grafică a aplicației

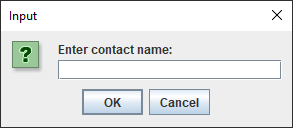


Figura 1.2 Crearea unui nou contact

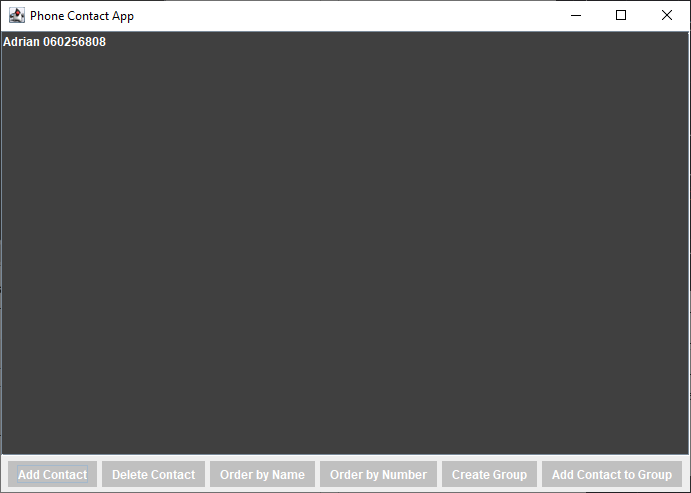


Figura 1.3 Exemplu de contact creat

# MODELAREA SISTEMULUI UML

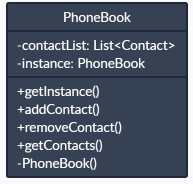


Figura 2.1 Singleton Pattern

Clasa PhoneBook reprezintă un manager de contacte în cadrul aplicației date. Aceasta toate atributele pattern-ului Singleton, astfel conține constructorul privat PhoneBook și metoda getInstance() care verifica daca această clasa a fost inițializată sau nu.

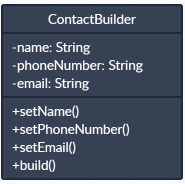


Figura 2.2 Builder Pattern

Clasa ContactBuilder este responsabilă pentru crearea obiectelor de tip Contact. Aceasta are în interior atributele clasei Contact si de asemenea metoda build() care instantiaza un obiect nou de tip Contact. Astfel crearea contactelor devine mai accesibila fara apelarea constructurilor clasei.

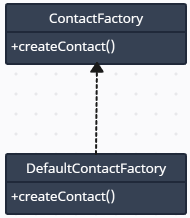


Figura 2.3 Factory Method

Interfața ContactFactory este responsabilă pentru crearea obiectelor de tip contact utilizănd diferite metode de creare, în cazul dat este prezentă metoda de baza pentru crearea contactelor (DefaultContactFactory). Astfel, ne asigurăm pe viitor că vom putea avea mai multe tipuri de creare a contactelor la necesitate.

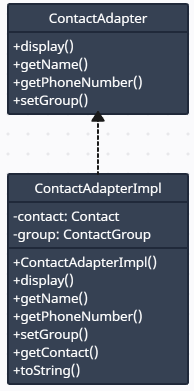


Figura 2.4 Adapter Pattern

Adapter Pattern este folosit pentru a adapta clasa existentă Contact la interfata ContactAdapter. Această interfață definește metodele getName() și getPhoneNumber(), care sunt așteptate de catre client pentru a face display si a manipula cu informația despre contacte. Clasa ContactAdapterImpl joaca rolul de adapter în acest scenariu.

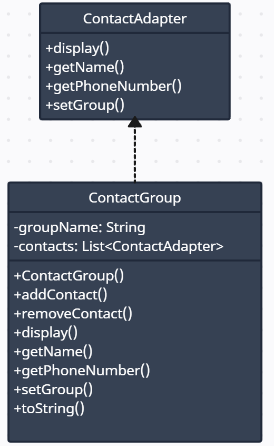


Figura 2.5 Composite Pattern

Composite pattern este folosit pentru a reprezenta relatiile intre contacte si grupuri de contacte. Clasa ContactGroup se comportă ca Composite, care poate conține atât contacte individuale, cât și alte grupe de contacte. ContactGroup implementează interfața ContactAdapter care permite acesteia să trateze contactele individuale si grupurile de contacte la fel.

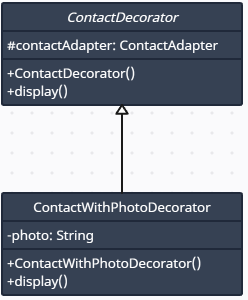


Figura 2.6 Decorator Pattern

Decorator Pattern a fost utilizat pentru a adăuga noi posibilități adiționale pentru contacte. ContactWithPhotoDecorator extinde funcționalitățile contactului și adaugă abilitatea de a face display la fotografia acestui contact. Acest decorator face wrap clasei ContactAdapter și adaugă funcționalitatea de a adauga fotografii.

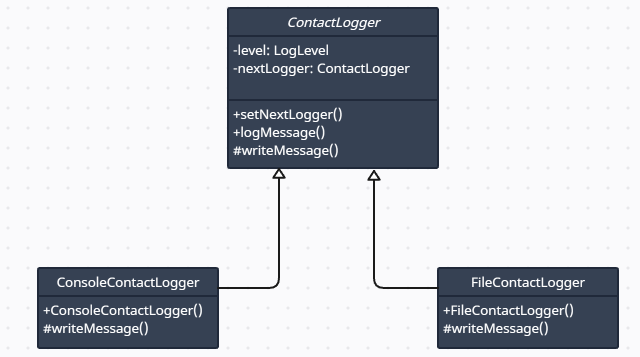


Figura 3.7 Chain of Responsability

Chain of Responsability pattern este folosit pentru a opera cu mesajele logate la diferite nivele (e.g., debug, info, warning, error) bazate pe severitatea acestora. Cand un mesaj este transmis catre ConactLogger, acesta incearca sa-l prelucreze. Daca loggerul poate sa-l prelucreze, atunci el intreprinde acțiunile necesare, dacă nu, atunci acesta transmite mesajul următorului logger in lanț.

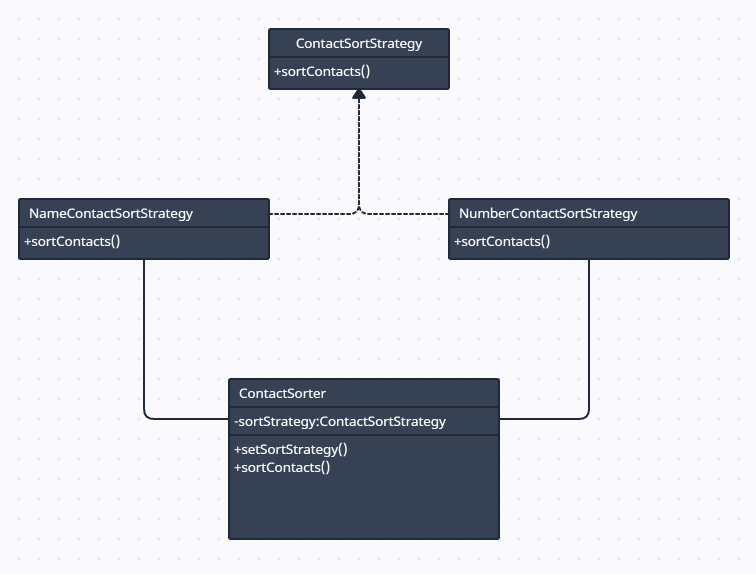


Figura 3.8 Strategy Pattern

ContactSorter este componenta care stabileste strategia de sortare a contactelor. Fiecare strategie reprezinta un algoritm specific de sortare precum ar fi sortarea dupa nume sau numar de telefon. ContactSortStrategy are referinta la strategia curenta de sortare care poate fi setata dinamic folosind setSortStrategy(). Cand metoda sort() este chemata, ea executa operatia de sortare adresandu-se la strategia curenta.

# CONCLUZIE

Aplicația de gestionare a contactelor este o soluție comprehensivă și flexibilă care folosește diverse modele de proiectare pentru a oferi capacități eficiente de organizare, sortare și jurnalizare a contactelor. Pe parcursul proiectului, au fost utilizate mai multe modele de proiectare pentru a aborda diferite aspecte ale funcționalității aplicației, promovând modularitatea, extensibilitatea și ușurința de întreținere.

Modelul Singleton a fost utilizat pentru a asigura existența unei singure instanțe a clasei "ContactManager", garantând astfel accesul centralizat la operațiile legate de contacte. Acest model îmbunătățește consistența datelor și evită supraîncărcarea inutilă a mai multor instanțe.

Modelul Builder a facilitat construcția obiectelor complexe "Contact" prin separarea procesului de construcție de crearea efectivă a obiectului. Utilizând "ContactBuilder", aplicația permite o modalitate flexibilă și lizibilă de a construi contacte cu diferite atribute, ceea ce face codul mai ușor de întreținut și reduce complexitatea constructorului.

Modelul Factory Method a fost utilizat pentru a încapsula crearea diferitelor tipuri de obiecte "Contact", cum ar fi "IndividualContact" și "OrganizationContact". Acest model permite o extindere ușoară și o personalizare a logicii de creare a contactelor, făcându-l scalabil și adaptabil la cerințele viitoare.

Modelul Adapter a fost utilizat pentru a adapta clasa existentă "Contact" la interfața "ContactAdapter". Acest lucru a permis tratarea uniformă a diferitelor tipuri de contacte, permițând integrarea fără probleme a diferitelor obiecte de contact în aplicație, fără modificarea codului existent.

Modelul Composite a fost utilizat pentru a reprezenta structura ierarhică a contactelor, în care contactele individuale și grupurile de contacte pot fi tratate în mod uniform. Acest model facilitează gestionarea și manipularea structurilor complexe de contacte, permițând o organizare ierarhică a contactelor și posibilitatea de a lucra cu grupuri și contacte individuale în mod interschimbabil.

Modelul Decorator a jucat un rol crucial în extinderea funcționalității contactelor în mod dinamic. Decoratorii precum "ContactDecorator" și "ContactWithPhotoDecorator" au adăugat funcții suplimentare la contactele de bază, cum ar fi afișarea numelor, numerelor de telefon și a fotografiilor. Acest model a permis personalizarea ușoară și extinderea obiectelor de contact fără modificarea clasei principale "Contact" sau introducerea ierarhiilor complexe de moștenire.

Modelul Chain of Responsibility a fost utilizat în componenta "ContactLogger" pentru a gestiona mesajele de jurnalizare cu diferite niveluri de gravitate. Acest model a permis decuplarea logică de gestionare a mesajelor de jurnalizare și a oferit flexibilitate în adăugarea sau modificarea loggerelor fără a afecta procesul general de jurnalizare.

Modelul Strategy a fost utilizat în componenta "ContactSorter" pentru a defini diferite strategii de sortare pentru contacte. Acest model a permis aplicației să selecteze și să schimbe dinamic algoritmul de sortare în funcție de cerințe specifice sau preferințele utilizatorului, promovând flexibilitatea și adaptabilitatea în sortarea contactelor.

În concluzie, aplicația de gestionare a contactelor demonstrează utilizarea eficientă a mai multor modele de proiectare pentru a crea un sistem robust și flexibil. Aplicația beneficiază de modelele Singleton, Builder, Factory Method, Adapter, Composite, Decorator, Chain of Responsibility și Strategy, care îmbunătățesc în mod colectiv modularitatea, extensibilitatea și ușurința de întreținere.

Aceste modele de proiectare oferă o bază solidă pentru îmbunătățiri și modificări viitoare ale aplicației. Natura modulară a designului permite adăugarea ușoară a unor noi funcționalități, cum ar fi tipuri suplimentare de contacte, strategii de sortare sau capacități de jurnalizare. De asemenea, permite aplicației să se adapteze la cerințele în schimbare și preferințele utilizatorilor fără a perturba codul existent.

Prin utilizarea acestor modele de proiectare, aplicația demonstrează puterea principiilor și a modelelor de proiectare software în crearea unei arhitecturi curate și scalabile. Separarea preocupărilor, încapsularea și flexibilitatea oferite de aceste modele contribuie la un cod ușor de înțeles, întreținut și extins.

În ansamblu, aplicația de gestionare a contactelor evidențiază implementarea reușită a mai multor modele de proiectare, rezultând un sistem robust, flexibil și ușor de utilizat. Aceasta servește ca o dovadă a importanței modelelor de proiectare în crearea unui software de înaltă calitate, care răspunde nevoilor utilizatorilor și poate evolua în timp.

# BIBLIOGRAFIE

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - "Design Patterns: Elements of Reusable Object-Oriented Software" (1994)

Gang of Four (GoF) Design Patterns - https://sourcemaking.com/design\_patterns

Christopher Alexander - "A Pattern Language: Towns, Buildings, Construction" (1977)

Martin Fowler - "Patterns of Enterprise Application Architecture" (2002)